下载APP

18 | 流量隔离：RabbitMQ 消息隔离是怎么做的？

2021-11-27 高楼

《全链路压测实战30讲》 课程介绍 >

讲述：高楼

时长 21:43 大小 19.89M

你好，我是高楼。

这节课，我们详细来讲讲如何基于微服务技术落地 RabbitMQ 消息隔离。

我们的项目中选择的消息中间件是 RabbitMQ。它是最受欢迎的开源消息中间件之一。RabbitMQ 量级轻，而且易于部署，能支持多种消息协议。它还可以部署在分布式系统中，满足大规模、高可用、削峰填谷的要求。所有消息中间件的根本目标是相同的，那就是：**将同步处理转成异步处理。**

首先，我们来了解一下 Spring 中 RabbitMQ 的消息传递解决方案。

Spring 生态中提供了Spring-AMQP 项目，可以让我们更简便地使用 AMQP。它提供了一个“template”作为发送消息的高级抽象。同时它还通过“Listener Container”为消息驱动的 POJO 提供支持。这些库简化了 AMQP 资源的开发使用。

这个项目主要包括两个部分：

- Spring-AMQP ：是 AMQP 的基础抽象；
- Spring-Rabbit ：是基于 RabbitMQ 对 AMQP 的具体实现。

它的主要功能包括：

- Listener Container：异步处理接收到的消息；
- RabbitTemplate ：发送和接收消息；
- RabbitAdmin ：自动创建队列、交换机、绑定器。

因为我们的项目采用 SpringBoot，而它默认提供了 AMQP 和 RabbitMQ 的自动化配置，所以我们仅需引入 spring-boot-starter-amqp 依赖，即可快速装配使用了。

消息隔离方案

刚才，我们简单介绍了 RabbitMQ，并了解了如何在 SpringBoot 中使用 RabbitMQ。接下来，我们回到这节课的主题：如何在 SpringBoot 中落地 MQ 消息隔离方案。

在这里我梳理了目前业界对于消息中间件隔离的主要解决方案，你可以参考下面的表格：

	数据偏移	影子队列
优点	• 实现简单，无须改造中间件	• 逻辑隔离 • 中间件容易支持 • 安全性高，对生产队列无任何影响 • 扩展性较强 • 数据清理简单
缺点	• 数据进入生产队列 • 对业务数据有侵入性，会影响消息内容 • 安全性较差，设计逻辑有纰漏，会影响生产数据 • 压力大的情况下，会挤占中间件的资源	• 压力大的情况下，会挤占中间件的资源
适用场景	• 流量标记没打通 • 技术体系不完整，改造成本和复杂度过高	• 全服务流量标记 • 中间件技术体系完整

极客时间

其中，数据偏移方式会让压测消息跟正常消息都进入到生产队列。压测消息在发送端加上标记，然后在接收端加上识别。

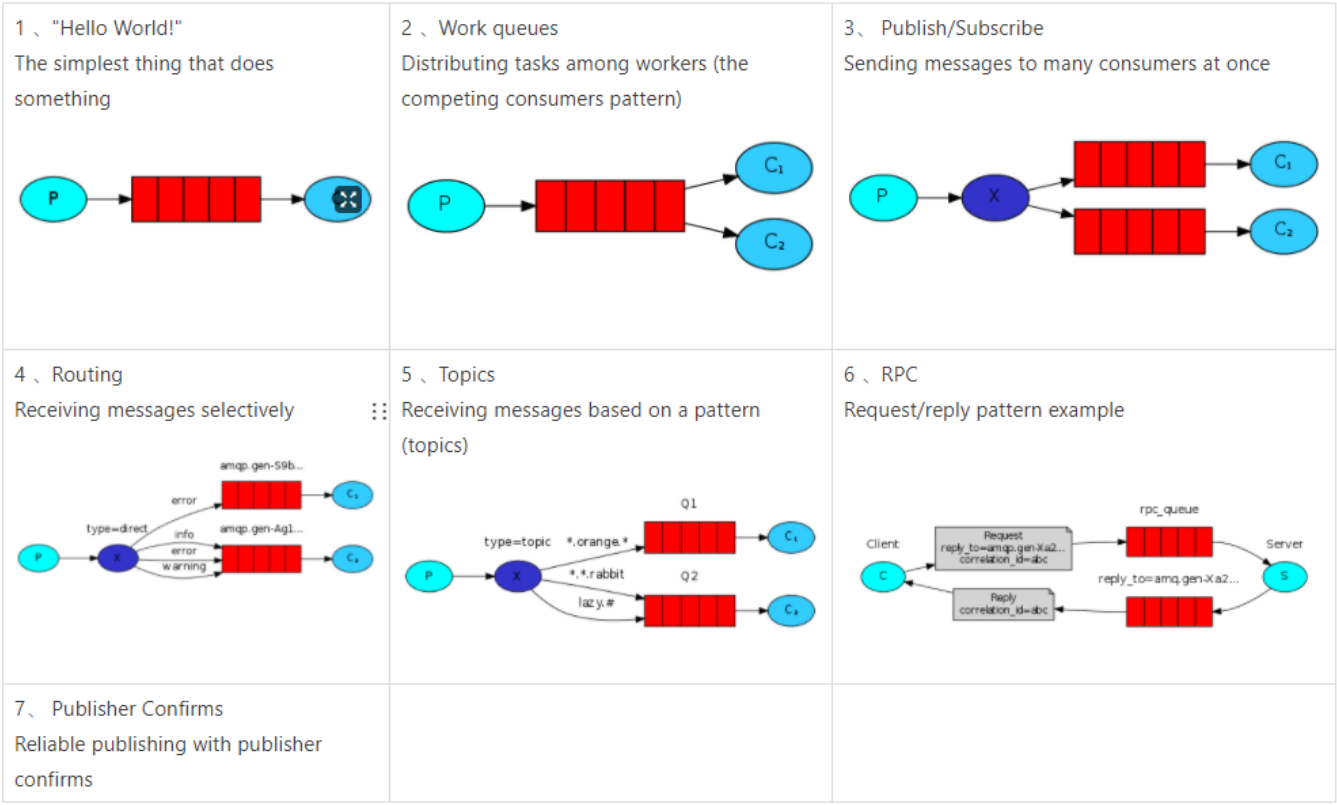
而影子队列方式则会把压测消息发送到另一个影子队列里，跟生产队列完全隔离开。接受端同时监听生产队列和影子队列，然后从接收的消息判断是哪个队列发过来的，再根据判断的结果做对应的处理。

根据不同的项目情况，我们可以选择不同的技术方案，当然，**这里我们首推影子队列，因为它相对安全**。

为了方便你参考，我会对影子队列和数据偏移方案分别进行演示。

技术预演

在正式预演之前，我们先要知道 RabbitMQ 消息模式的实现方式，RabbitMQ 的 7 种消息模式是构建消息应用的基础，我们一定要牢牢掌握它们。



学过 RabbitMQ 的朋友应该了解过这些消息模式的 Java 实现方式，这里的 demo 预演我们使用 Spring AMQP 的形式来实现它们。

在 AMQP 中，发布者将消息发送到交换机，再由交换机将消息路由到一个或多个消息队列中（或者丢弃）。

交换机会根据路由键和绑定的键将消息路由到消息队列。目前常用的交换器类型有 Direct、Topic、Fanout、Headers 四种类型。

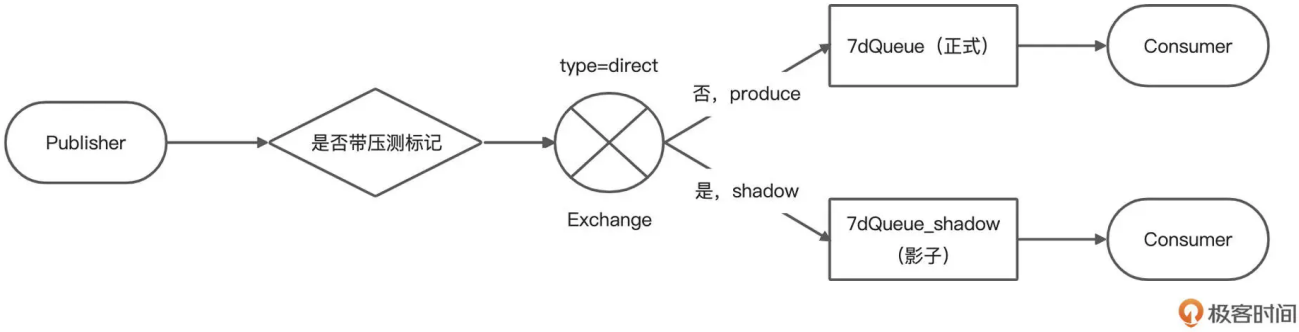
影子队列实现

我们先来看一下影子队列的实现方式。

影子队列我们主要通过 RabbitMQ 的路由模式（Direct）实现：

> 路由模式是可以根据路由键选择性给多个消费者发送消息的模式。它包含一个发布者、两个消费者、两个队列和一个交换机。两个消费者同时绑定到不同的队列上去，两个队列通过路由键绑定到交换机上去，发布者发送消息到交换机，交换机通过路由键转发到不同队列，队列绑定的消费者接收并消费消息。

为了方便你更直观地理解路由模式的原理，我给你画了一张示意图。



通过这张图我们可以很直观的看到：

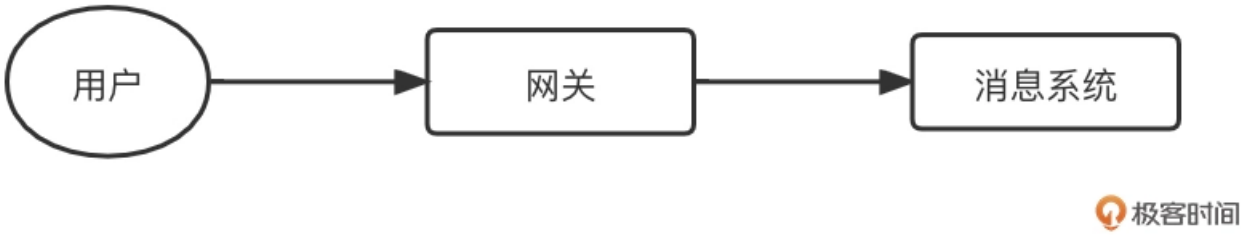
如果我们以 routingKey=“produce” 发送消息到交换机，消息会路由到 7dQueue。

如果我们以 routingKey=“shadow” 发送消息到交换机，消息会路由到 7dQueue_shadow。

也就是说，如果我们在交换机设置对应路由键，那么消息就只会路由到最多一个消息队列中。如果没有相应的匹配，消息就会发送到“空气”（丢弃）中，不会进入任何消息队列中了。

我们还是来看一下具体的操作步骤吧。

这里使用的 demo 工程还是 14 讲的示例，主要包括网关和消息系统 2 个服务：



下面我们主要改造消息系统，实现 RabbitMQ 消息隔离技术预演。

第一步，引入相关依赖。

复制代码

```
<dependencies>
    <dependency>
        <groupId>com.dunshan</groupId>
        <artifactId>dunshan-common</artifactId>
        <version>1.0-SNAPSHOT</version>
        <scope>compile</scope>
        <exclusions>
            <exclusion>
                <artifactId>jsr305</artifactId>
                <groupId>com.google.code.findbugs</groupId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-aop</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <!--Swagger-UI API 文档生产工具-->
    <dependency>
        <groupId>io.springfox</groupId>
        <artifactId>springfox-swagger2</artifactId>
        <version>2.7.0</version>
    </dependency>
    <dependency>
        <groupId>io.springfox</groupId>
        <artifactId>springfox-swagger-ui</artifactId>
        <version>2.7.0</version>
    </dependency>
    <!--hutool 工具包-->
    <dependency>
        <groupId>cn.hutool</groupId>
        <artifactId>hutool-all</artifactId>
        <version>4.6.3</version>
    </dependency>
    <!--Spring AMQP 依赖-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-amqp</artifactId>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.10</version>
        <scope>provided</scope>
    </dependency>
```

```
51 </dependencies>
```

第二步，在 resources 目录下，创建 application.yaml 配置文件。

复制代码

```
1  server:
2    port: 8008
3
4  spring:
5    rabbitmq:
6      host: localhost
7      port: 5672
8      virtual-host: /
9      username: guest
10     password: guest
11     publisher-confirms: true #消息发送到交换器确认
12     publisher-returns: true #消息发送到队列确认
```

这里，我们在 spring.rabbitmq 配置项，设置了 RabbitMQ 的配置，对应的是 RabbitProperties 配置类。

然后，我们可以利用 SpringBoot 提供的 RabbitAutoConfiguration 自动化配置类，实现 RabbitMQ 的自动配置，创建相应的发布者和消费者。

第三步，创建 RabbitConfig 配置类，添加和 Direct Exchange 相关的 Exchange、Queue、Binding 配置。

这一步主要是添加和路由模式相关 Java 配置，创建一个名为 exchange.7d 的交换机、一个发布者、两个消费者和两个消息队列。其中，队列通过路由键都绑定到交换机。

消息队列 7dQueue 的路由键为 produce；

消息队列 7dQueue_shadow 的路由键为 shadow。

复制代码

```
1  package com.dunshan.direct;
2
3  import org.springframework.amqp.core.Binding;
4  import org.springframework.amqp.core.BindingBuilder;
5  import org.springframework.amqp.core.DirectExchange;
```

```
import org.springframework.amqp.core.Queue;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

/**
 *
 * @author dunshan
 * @date 2021/10/1
 */
@Configuration
public class DirectRabbitConfig {

    @Bean
    public DirectExchange direct() {
        return new DirectExchange("exchange.7d");
    }

    @Bean
    public Queue directQueue() {
        return new Queue("7dQueue");
    }

    @Bean
    public Queue directQueueShadow() {
        return new Queue("7dQueue_shadow");
    }
    @Bean
    public Binding directBinding1a(DirectExchange direct, Queue directQueue) {
        return BindingBuilder.bind(directQueue).to(direct).with("produce");
    }

    @Bean
    public Binding directBinding1b(DirectExchange direct, Queue directQueueSha
        return BindingBuilder.bind(directQueueShadow).to(direct).with("shadow"
    }

    @Bean
    public DirectReceiver receiver() {
        return new DirectReceiver();
    }

    @Bean
    public DirectSender directSender() {
        return new DirectSender();
    }
}
```

从这段代码中可以看到，在 DirectExchangeDemoConfiguration 内部静态类中，我们创建了 Exchange、Queue、Binding 三个 Bean，后续 RabbitAdmin 会自动创建交换机、

消息队列、绑定器。

第四步，实现获取压测标记。

接下来我们就要考虑如何获取压测请求 Header 标记了。

首先还是实现一个全局 Filter 过滤器，只要从 Span Baggage 获取压测标记放入 Context（数据上下文），就能在服务中使用了。

全局 Filter 过滤器代码如下：

复制代码

```
package com.dunshan.config;

import brave.Span;
import brave.Tracer;
import brave.baggage.BaggageField;
import lombok.extern.log4j.Log4j2;
import org.apache.commons.lang3.StringUtils;
import org.springframework.stereotype.Component;
import javax.servlet.*;
import java.io.IOException;

/**
 * @author dunshan
 * @description: 全局过滤器
 * @date 2021-10-03 17:45:30
 */
@Log4j2
@Component
public class ContextFilter implements Filter {

    private final Tracer tracer;
    ContextFilter(Tracer tracer) {
        this.tracer = tracer;
    }

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        Filter.super.init(filterConfig);
    }

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servle
        BaggageField dunshan = BaggageField.getByName("dunshan");
        String flag = dunshan.getValue();
```

```
35          Span currentSpan = this.tracer.currentSpan();
36          AppContext appContext = new AppContext();
37          if (StringUtils.isNotBlank(flag)) {
38              log.info("压测流量: " + flag);
39              currentSpan.tag("flag", flag);
40              appContext.setFlag(flag);
41              AppContext.setContext(appContext);
42          } else {
43              log.info("正常流量: " + flag);
44              AppContext.setContext(appContext);
45          }
46          filterChain.doFilter(servletRequest, servletResponse);
47      }
48
49      @Override
50      public void destroy() {
51          AppContext.removeContext();
52          Filter.super.destroy();
53      }
54
55  }
```

数据上下文代码如下：

复制代码

```
1  package com.dunshan.config;
2  import com.alibaba.ttl.TransmittableThreadLocal;
3  import java.io.Serializable;
4
5  /**
6   * @author dunshan
7   * @description: 数据上下文
8   * @date 2021-11-12 17:53:39
9   */
10 public class AppContext implements Serializable {
11
12     private static final TransmittableThreadLocal<AppContext> contextdunshan =
13     private String flag;
14
15     public static AppContext getContext() {
16         return contextdunshan.get();
17     }
18
19     public static void setContext(AppContext context) {
20         contextdunshan.set(context);
21     }
22
23     public static void removeContext() {
24         contextdunshan.remove();
```

```
    }

    public String getFlag() {
        return flag;
    }

    public void setFlag(String flag) {
        this.flag = flag;
    }

}
```

后面，我们只要在业务方通过从 getContext().getFlag() 中取出 Header 标记即可。

代码如下：

复制代码

```
// 获取标记
String header = AppContext.getContext().getFlag();
```

第五步，创建 DirectSender 类。

它会使用 Spring-AMQP 封装提供的 RabbitTemplate 来发送消息。发布者通过 send 方法向交换机 exchange.7d 发送消息。交换机通过判断是否为压测标记，来选择使用不同的路由键，然后，这些消息会根据不同的路由键被转发到不同的队列。

复制代码

```
package com.dunshan.direct;

import com.dunshan.config.AppContext;
import lombok.extern.slf4j.Slf4j;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.beans.factory.annotation.Autowired;

/**
 *
 * @author dunshan
 * @date 2021/10/1
 */
@Slf4j
public class DirectSender {

```

```
    @Autowired
    private RabbitTemplate template;

    private static final String exchangeName = "exchange.7d";

    public void send(int index) {
        // 获取标记
        String header = AppContext.getContext().getFlag();
        StringBuilder builder = new StringBuilder("Hello to ");
        String message = builder.toString();

         // 判断标记
        if ("7DGroup".equals(header)) {
            template.convertAndSend(exchangeName, "shadow", message);
        } else {
            template.convertAndSend(exchangeName, "produce", message);
        }
        log.info(" [x] Sent '{}'", message);
    }

}
```

在具体的代码实现上，因为 RabbitTemplate 是 AmqpTemplate 接口的实现类，所以使用 AmqpTemplate 也可以。不过因为 RabbitTemplate 还实现了其它接口，操作会更为丰富一些。所以这里我们还是选择注入了 RabbitTemplate 属性。

第六步，创建 DirectReceiver 类，消费消息。

消费者从自己绑定的匿名队列中获取消息，由于该消费者可以从两个队列中获取并消费消息，所以可以看做是两个消费者，名称分别是 instance 1 和 instance 2。

复制代码

```
package com.dunshan.direct;

import cn.hutool.core.thread.ThreadUtil;
import lombok.extern.slf4j.Slf4j;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.util.StopWatch;

/**
 *
 * @author dunshan
 * @date 2021/10/1
 */
@Slf4j
public class DirectReceiver {
```

```
15
16      @RabbitListener(queues = "#{directQueue.name}")
17      public void receive1(String in){
18          receive(in, 1);
19      }
20
21      @RabbitListener(queues = "#{directQueueShadow.name}")
22      public void receive2(String in){
23          receive(in, 2);
24      }
25
26      private void receive(String in, int receiver){
27          StopWatch watch = new StopWatch();
28          watch.start();
29          log.info("instance {} [x] Received '{}'", receiver, in);
30          doWork(in);
31          watch.stop();
32          log.info("instance {} [x] Done in {}s", receiver, watch.getTotalTimeSe
33      }
34
35      private void doWork(String in){
36          for (char ch : in.toCharArray()) {
37              if (ch == '.') {
38                  ThreadUtil.sleep(1000);
39              }
40          }
41      }
42
43  }
```

可以看到，我们在类上添加了 @RabbitListener 注解，声明了消费的队列。

第七步，在 Controller 中添加测试接口。

这里主要实现调用该接口，开始发送消息。

复制代码

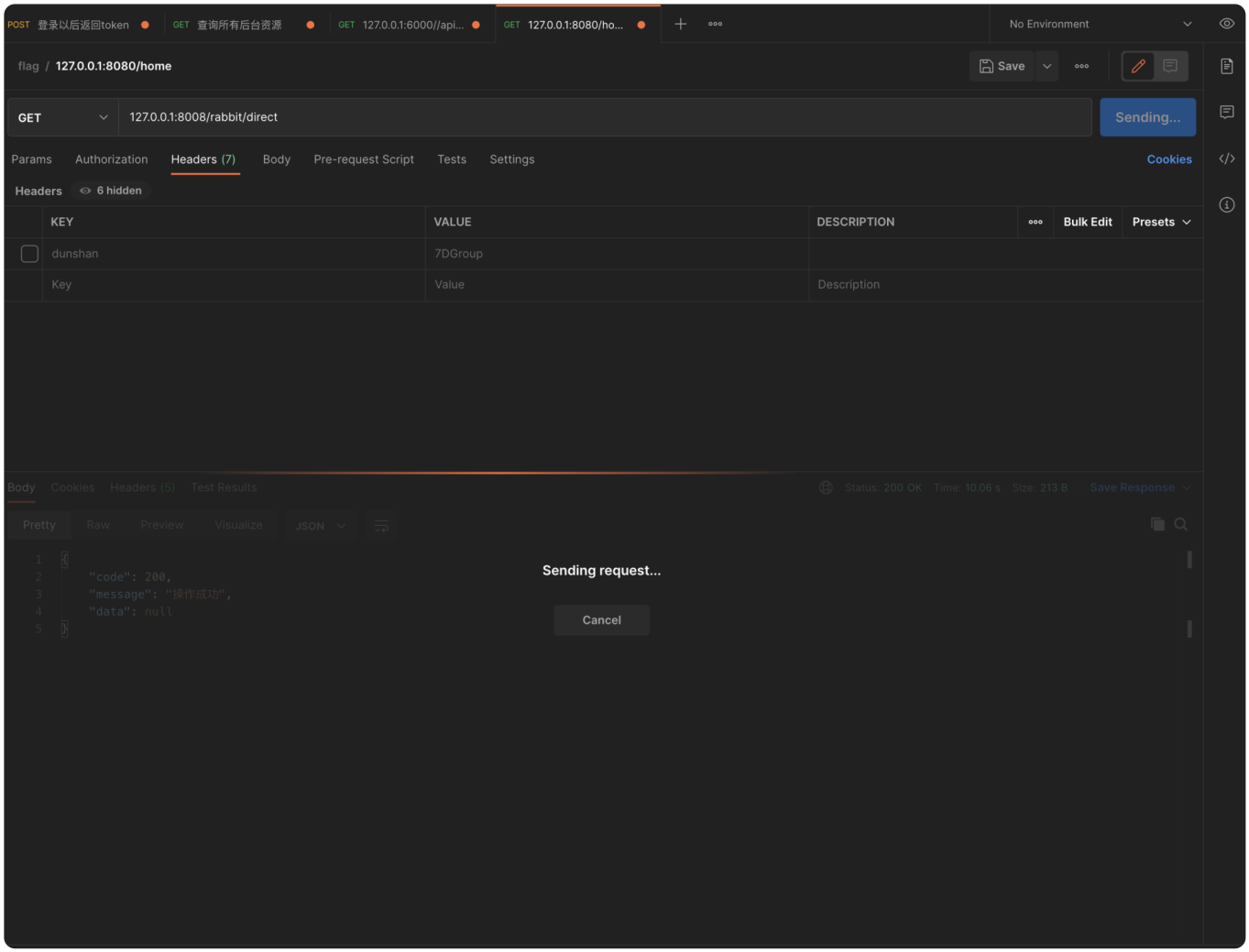
```
1  package com.dunshan.controller;
2
3  import cn.hutool.core.thread.ThreadUtil;
4  import com.dunshan.common.api.CommonResult;
5  import com.dunshan.direct.DirectSender;
6  import com.dunshan.directheader.DirectHeaderSender;
7  import io.swagger.annotations.Api;
8  import io.swagger.annotations.ApiOperation;
9  import org.slf4j.Logger;
10 import org.slf4j.LoggerFactory;
```

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;

/**
 * Created by dunshan on 2020/5/19.
 */
@Api(tags = "RabbitController", description = "RabbitMQ 消息隔离测试")
@Controller
@RequestMapping("/rabbit")
@Slf4j
public class RabbitController {

    @Autowired
    private DirectSender directSender;

    @ApiOperation("路由模式")
    @RequestMapping(value = "/direct", method = RequestMethod.GET)
    @ResponseBody
    public CommonResult directTest() {
        for(int i=0;i<10;i++){
            directSender.send(i);
            ThreadUtil.sleep(1000);
        }
        return CommonResult.success(null);
    }

}
```

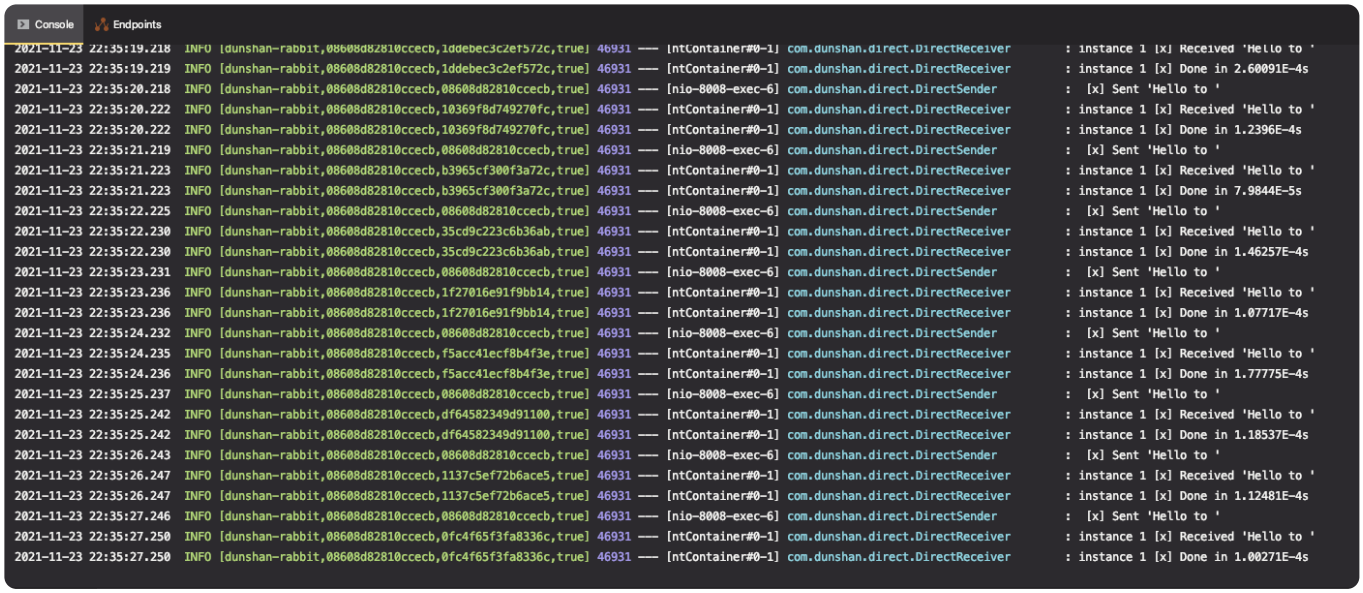
第八步，接口测试验证。

我们可以通过 Postman 做接口测试，往队列中发送包含不同路由键的消息。

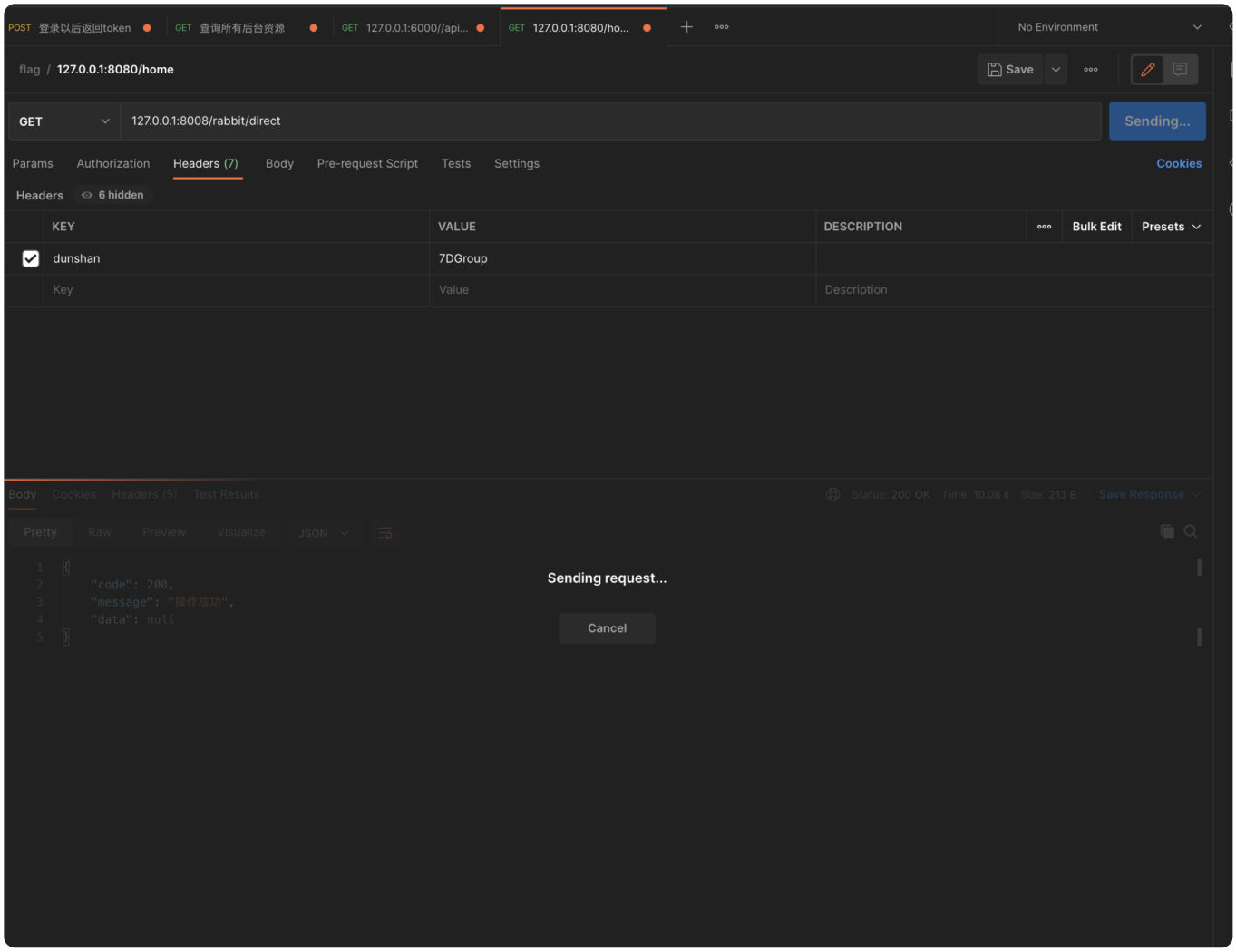
发送正常请求（不带压测标记）



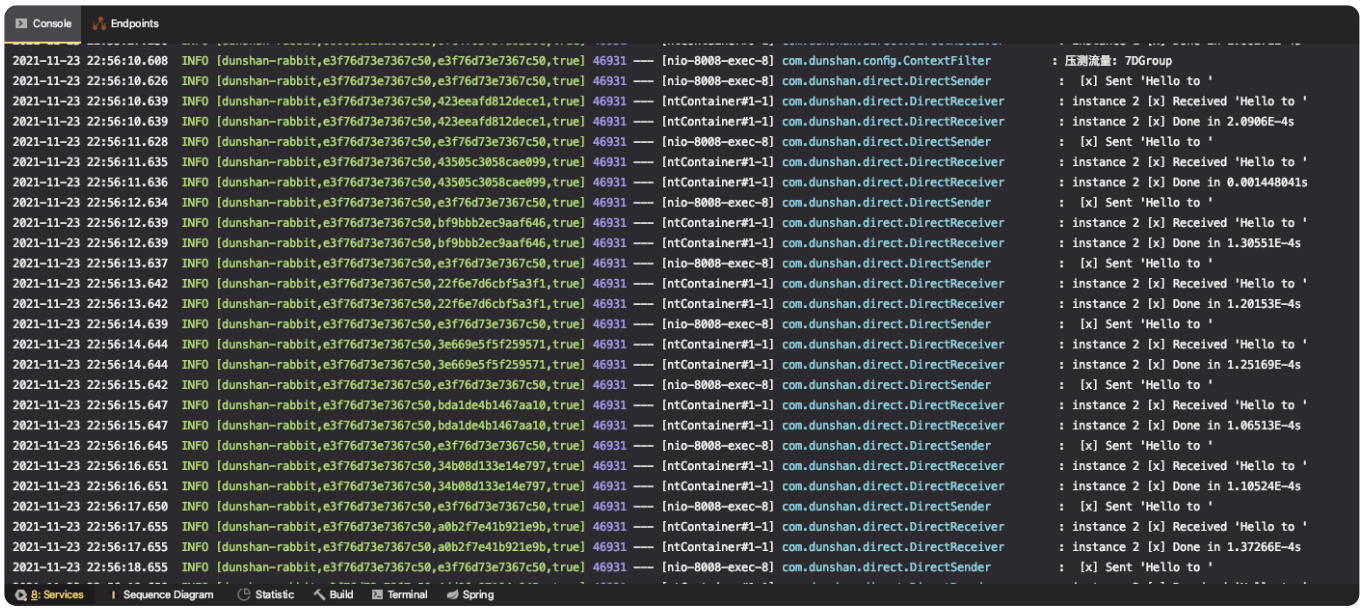
服务端的控制台日志如下：



发送压测请求（带压测标记）



服务端的控制台日志如下：



可以看到，instance 1 获取到了正常消息，instance 2 获取到了压测消息，这说明测试成功了。

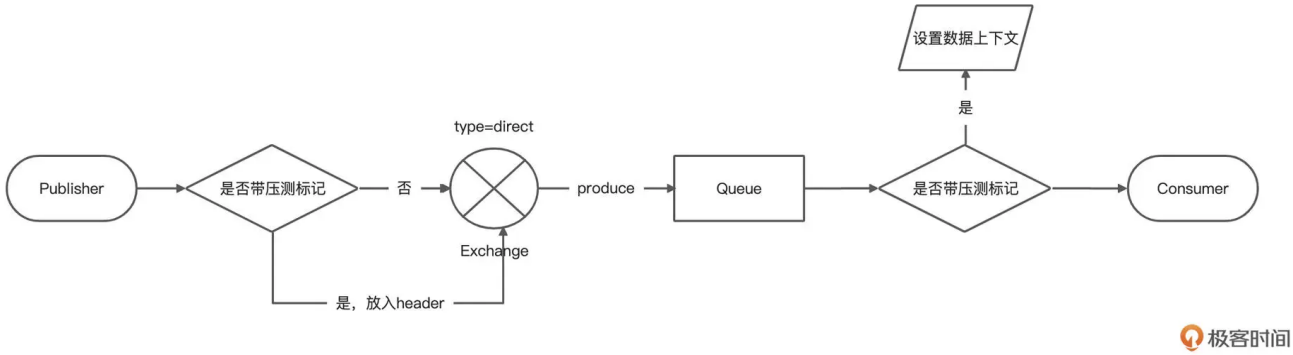
这样的话，后续业务我们就可以判断消息是不是从影子队列发过来的了。我们还可以将压测标记设置到数据上下文，最后根据这个标记来写数据库或进行其他操作。

数据偏移实现

好了，接下来，我们再来看下数据偏移的实现方式。

数据偏移方式会让压测请求和正常请求都进入到生产队列。我们要在压测消息的发布端加上标记，消费端加上识别。

你可以通过下面这张图片更直观地理解数据偏移的工作原理。



第一步，改造 RabbitConfig 配置类。

在影子队列方案的基础上，我们还需要改造 RabbitConfig 配置类，创建一个名为 exchange.7d 的交换机、一个生产者、两个消费者和一个消息队列。队列需要通过路由键绑定到交换机，其中队列 7dQueue 的路由键为 produce。

因为消息的标记识别实际上就是对发布和消费做一个拦截处理，所以我们还要配置 1 个 Bean，在 Bean 的方法里面实现拦截的逻辑。

复制代码

```
1 package com.dunshan.directheader;
2
3 import lombok.extern.slf4j.Slf4j;
4 import org.springframework.amqp.AmqpException;
5 import org.springframework.amqp.core.*;
6 import org.springframework.amqp.rabbit.config.SimpleRabbitListenerContainerFac
7 import org.springframework.amqp.rabbit.connection.CachingConnectionFactory;
8 import org.springframework.amqp.rabbit.connection.ConnectionFactory;
```

```
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.boot.autoconfigure.amqp.SimpleRabbitListenerContain
import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.context.request.RequestContextHolder;
import org.springframework.web.context.request.ServletRequestAttributes;

import javax.servlet.http.HttpServletRequest;
import java.util.Map;

/**
 * @author dunshan
 */



@Configuration
@Slf4j
public class DirectRabbitHeaderConfig {

    @Bean
    public DirectExchange directheader() {
        return new DirectExchange("exchange.7dHeader");
    }

    @Bean
    public Queue directheaderQueue() {
        return new Queue("7dHeaderQueue");
    }

    @Bean
    public Binding directheaderBinding1a(DirectExchange directheader, Queue di
        return BindingBuilder.bind(directheaderQueue).to(directheader).with("p
    }



    @Bean
    public DirectHeaderReceiver headerreceiver() {
        return new DirectHeaderReceiver();
    }



    @Bean
    public DirectHeaderSender directHeaderSender() {
        return new DirectHeaderSender();
    }

    @Bean(name = "rabbitListenerContainerFactory")
    @ConditionalOnProperty(prefix = "spring.rabbitmq.listener", name = "type",
```

```
61      public SimpleRabbitListenerContainerFactory simpleRabbitListenerContainerF
62
63          SimpleRabbitListenerContainerFactory factory = new SimpleRabbitListene
64          factory.setAfterReceivePostProcessors(new MessagePostProcessor() {
65              @Override
66              public Message postProcessMessage(Message message) throws AmqpExce
67                  Map header = message.getMessageProperties().getHeaders();
68                  //判断是否压测消息，是的话要动态切换影子库跟后续操作
69                  if (header.containsKey("test")){
70                      log.info("带压测标记！");
71                  }
72                  return message;
73              }
74          });
75          configurer.configure(factory, connectionFactory);
76          return factory;
77      }
78
79  }
```

每次接收消息都会调用拦截器对消息进行拦截处理，有压测消息标记的，就先存到数据上下文中，后续数据库会根据这个上下文切换影子库。

第二步，改造 DirectSender 类。

然后改造 DirectSender 类，发布消息时判断是否为压测标记，使用 MessageProperties 构建自定义 Header，以达到数据偏移的目的。

复制代码

```
1  package com.dunshan.directheader;
2
3  import lombok.extern.slf4j.Slf4j;
4  import org.springframework.amqp.core.Message;
5  import org.springframework.amqp.core.MessageProperties;
6  import org.springframework.amqp.rabbit.core.RabbitTemplate;
7  import org.springframework.beans.factory.annotation.Autowired;
8  import org.springframework.web.context.request.RequestContextHolder;
9  import org.springframework.web.context.request.ServletRequestAttributes;
10
11 import javax.servlet.http.HttpServletRequest;
12
13 /**
14  *
15  * @author dunshan
16  * @date 2021/10/1
17  */
```

```
@Slf4j
public class DirectHeaderSender {

  @Autowired
  private RabbitTemplate template;

  private static final String exchangeName = "exchange.7dHeader";
  private final String key = "produce";

  public void send(int index) {

    // 获取标记
    String header = AppContext.getContext().getFlag();

    String messageStr = "Hello to ";

    MessageProperties messageProperties = new MessageProperties();
    // 判断标记
    if (header != null && "7DGroup".equals(header)) {
      messageProperties.setHeader("test",true);
      Message message = new Message(messageStr.getBytes(), messageProperties);
      template.convertAndSend(exchangeName, key, message);
      log.info(" [x] Sent '{}'", message);
    } else {
      template.convertAndSend(exchangeName, key, messageStr);
      log.info(" [x] Sent '{}'", messageStr);
    }

  }

}
```

第三步，在 Controller 中添加测试接口。

在 Controller 中添加测试接口，调用该接口，发布消息。

复制代码

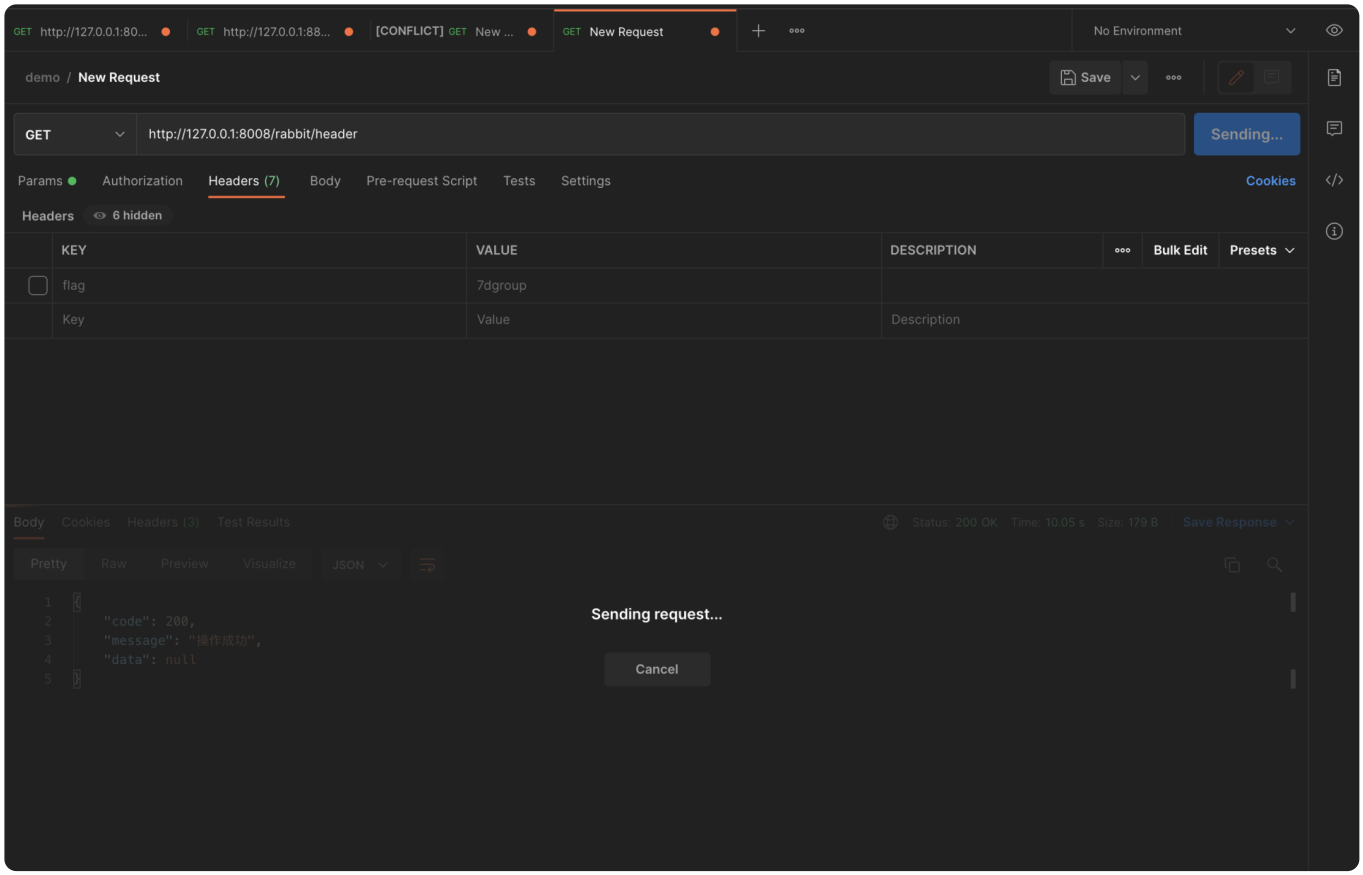
```
package com.dunshan.controller;

import cn.hutool.core.thread.ThreadUtil;
import com.dunshan.common.api.CommonResult;
import com.dunshan.direct.DirectSender;
import io.swagger.annotations.Api;
import io.swagger.annotations.ApiOperation;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
```

```
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;

/**
 *
 * @author dunshan
 * @date 2021/10/1
 */

@Api(tags = "RabbitController", description = "RabbitMQ 隔离测试")
@Controller
@RequestMapping("/rabbit")
public class RabbitController {

    @Autowired
    private DirectSender directSender;

    @ApiOperation("数据偏移模式")
    @RequestMapping(value = "/header", method = RequestMethod.GET)
    @ResponseBody
    public CommonResult headerTest() throws InterruptedException {

        for(int i=0;i<10;i++){
            directHeaderSender.send(i);

            ThreadUtil.sleep(1000);
        }
        return CommonResult.success(null);
    }

}
```

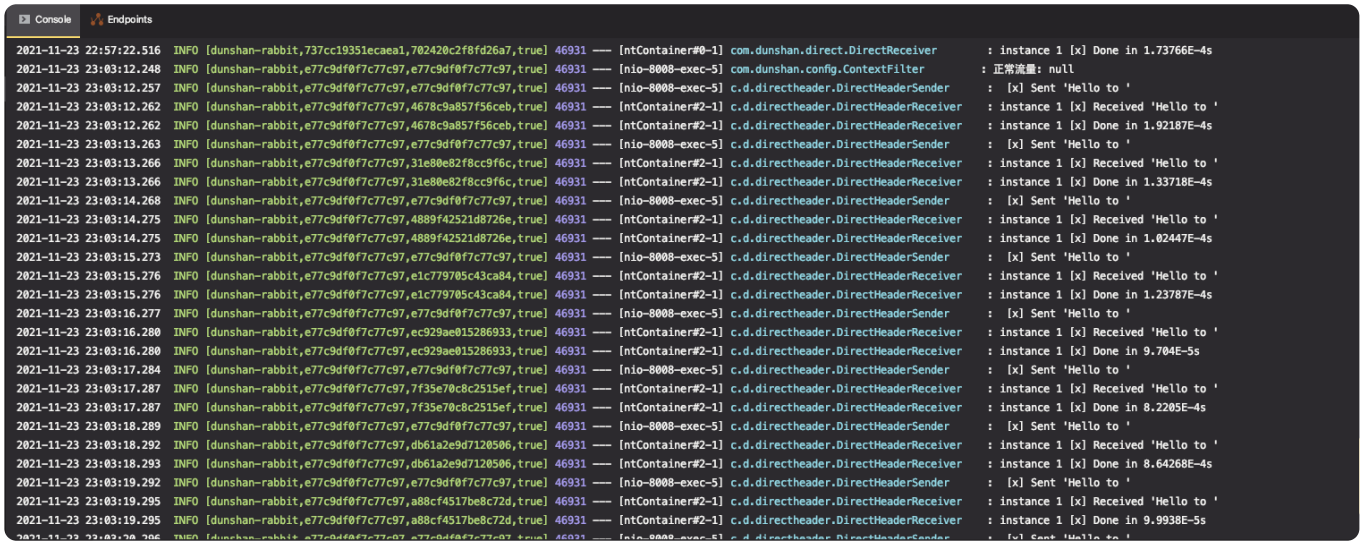
第四步，接口测试验证。

运行后，可以通过 Postman 做接口测试，往队列中发送包含不同路由键的消息。

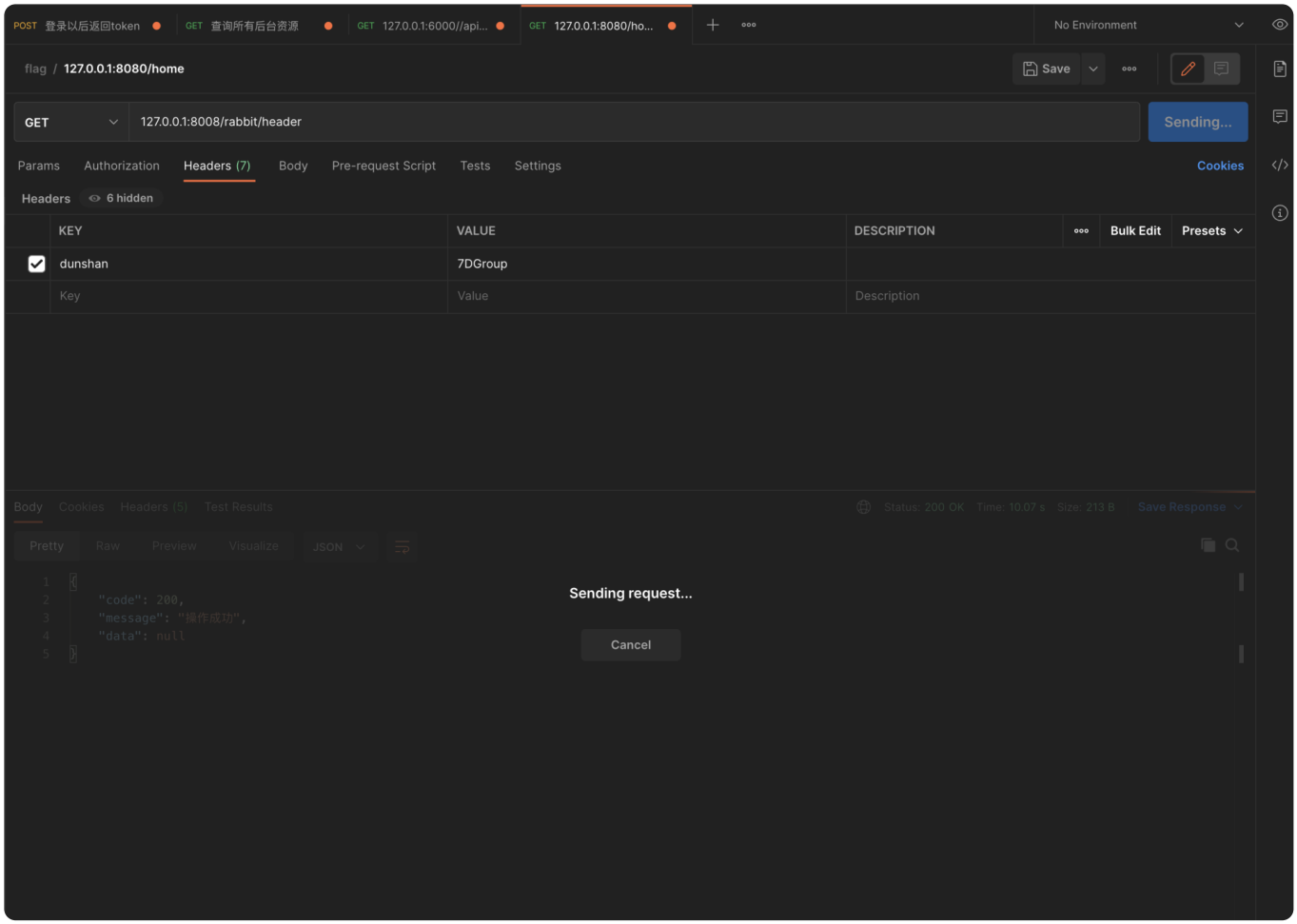
发送正常请求（不带压测标记）



服务端的控制台日志如下：



发送压测请求（带压测标记）



服务端的控制台日志如下：

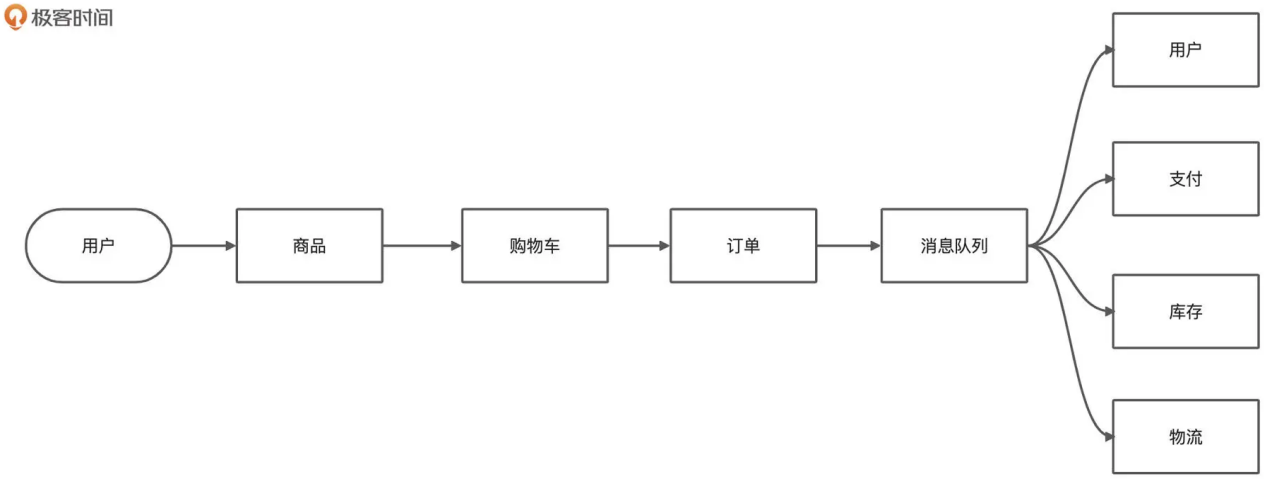
我们可以看到，instance 1 第一次获取到了正常消息，第二次获取到了压测标记的消息，这说明测试成功了。

刚才我们介绍了影子队列和数据偏移两种数据隔离方案并分别进行了 demo 预演。预演成功，接下来，我们就要将 demo 移植到真实系统进行改造了。

真实系统改造

在进行改造之前，先来看下我们项目的实际情况。

我们项目的消息队列主要是用于解决用户下单以后，订单超时想要取消订单的问题的。



项目主要的业务流程是：

用户进行下单操作（会有锁定商品库存、使用优惠券、积分一系列的操作）；

生成订单，获取订单的 id；

获取到设置的订单超时时间（假设设置的为 60 分钟不支付取消订单）；

按订单超时时间发送一个延迟消息给 RabbitMQ，让它在订单超时后触发取消订单的操作；

如果用户没有支付，进行取消订单操作（释放锁定商品库存、返还优惠券、返回积分一系列操作）。

RabbitMQ 实现延迟消息的方式有两种，一种是用死信队列实现，另一种是用延迟插件实现，我们这个项目使用的是死信队列的方式。

查看添加消息队列的枚举配置类 QueueEnum。

复制代码

```
1 import lombok.Getter;
2
3 /**
```

```
 * 消息队列枚举配置
 * Created by dunshan on 2018/9/14.
 */
@Getter
public enum QueueEnum {
    /**
     * 消息通知队列
     */
    QUEUE_ORDER_CANCEL("mall.order.direct", "mall.order.cancel", "mall.order.c

    /**
     * 消息通知 ttl 队列
     */
    QUEUE_TTL_ORDER_CANCEL("mall.order.direct.ttl", "mall.order.cancel.ttl", "

    /**
     * 交换名称
     */
    private String exchange;

     /**
     * 队列名称
     */
    private String name;

    /**
     * 路由键
     */
    private String routeKey;

    QueueEnum(String exchange, String name, String routeKey) {
        this.exchange = exchange;
        this.name = name;
        this.routeKey = routeKey;
    }
}
```

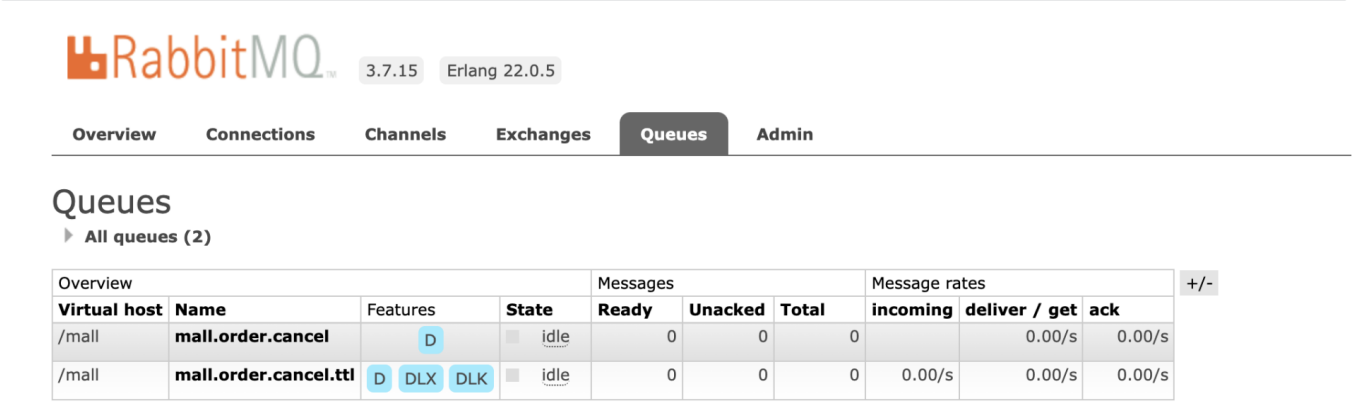
这里，我们定义了两个消息队列，即 mall.order.cancel 和 mall.order.cancel.ttl。

mall.order.direct（取消订单消息队列所绑定的交换机）：绑定的队列为 mall.order.cancel，一旦有消息以 mall.order.cancel 为路由键发过来，就会发送到这个队列。

mall.order.direct.ttl（订单延迟消息队列所绑定的交换机）：绑定的队列为 mall.order.cancel.ttl，一旦有消息以 mall.order.cancel.ttl 为路由键发送过来，就会转

发到此队列，并在此队列保存一定时间，等到超时后会自动将消息发送到 mall.order.cancel（取消订单消息消费队列）。

启动服务后，我们去 RabbitMQ 控制台确认一下。



这里我们选择的改造方案是使用**数据偏移**，因为相对影子队列的方式来说，数据偏移要改造的代码更少。

第一步，移植获取标记类。

快速移植 demo 中的全局 Filter 过滤器和数据上下文类：

AppContext；

ContextFilter。

这个操作前面说过很多次了，这里就不再多赘述了。

第二步，改造 CancelOrderSender 类。

改造 CancelOrderSender 类，发送时判断是否为压测标记，使用 MessageProperties 构建自定义 Header。

复制代码

```
1 package com.dunshan.mall.order.component;
2
3 import com.dunshan.mall.order.domain.QueueEnum;
4 import com.dunshan.mall.order.config.AppContext;
5 import lombok.extern.slf4j.Slf4j;
```

```java
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.amqp.AmqpException;
import org.springframework.amqp.core.AmqpTemplate;
import org.springframework.amqp.core.Message;
import org.springframework.amqp.core.MessagePostProcessor;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.web.context.request.RequestContextHolder;
import org.springframework.web.context.request.ServletRequestAttributes;

import javax.servlet.http.HttpServletRequest;

/**
 * 取消订单消息的发出者
 * Created by dunshan on 2018/9/14.
 */
@Component
@Slf4j
public class CancelOrderSender {
    private static Logger LOGGER =LoggerFactory.getLogger(CancelOrderSender.cl

    @Autowired
    private AmqpTemplate amqpTemplate;

    public void sendMessage(Long orderId,final long delayTimes){

        // 获取标记
        String header = AppContext.getContext().getFlag();

        LOGGER.info("RabbitMQ 获取标记 header:{}",header);

        //给延迟队列发送消息
        amqpTemplate.convertAndSend(QueueEnum.QUEUE_TTL_ORDER_CANCEL.getExchan

            @Override
            public Message postProcessMessage(Message message) throws AmqpExce

                if ("7DGroup".equals(header)) {
                    message.getMessageProperties().setHeader("test",true);
                }

                //给消息设置延迟毫秒值
                message.getMessageProperties().setExpiration(String.valueOf(de
                return message;
            }
        });
        LOGGER.info("send orderId:{}",orderId);
    }
}
```

第三步，改造 RabbitMqConfig 类。

然后，改造 RabbitMqConfig 配置类。配置 1 个 Bean，在 Bean 的方法里面实现拦截的逻辑。

每次接收消息都会调用拦截器对消息进行拦截处理，有压测消息标记的，就先存到数据上下文中，后续数据库会根据这个上下文切换影子库。

复制代码

```
package com.dunshan.mall.order.config;

import com.dunshan.mall.order.config.AppContext;
import com.dunshan.mall.order.domain.QueueEnum;
import lombok.extern.slf4j.Slf4j;
import org.springframework.amqp.AmqpException;
import org.springframework.amqp.core.*;
import org.springframework.amqp.rabbit.config.SimpleRabbitListenerContainerFac
import org.springframework.amqp.rabbit.connection.ConnectionFactory;
import org.springframework.boot.autoconfigure.amqp.SimpleRabbitListenerContain
import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.util.Map;

/**
 * 消息队列配置
 * Created by dunshan on 2018/9/14.
 */

@Configuration
@Slf4j
public class RabbitMqConfig {

    /**
     * 订单消息实际消费队列所绑定的交换机
     */
    @Bean
    DirectExchange orderDirect() {
        return (DirectExchange) ExchangeBuilder
                .directExchange(QueueEnum.QUEUE_ORDER_CANCEL.getExchange())
                .durable(true)
                .build();
    }

    /**
     * 订单延迟队列队列所绑定的交换机
```

```
     */
    @Bean
    DirectExchange orderTtlDirect() {
        return (DirectExchange) ExchangeBuilder
                .directExchange(QueueEnum.QUEUE_TTL_ORDER_CANCEL.getExchange()
                .build();
    }

    /**
     * 订单实际消费队列
     */
    @Bean
    public Queue orderQueue() {
        return new Queue(QueueEnum.QUEUE_ORDER_CANCEL.getName());
    }

    /**
     * 订单延迟队列（死信队列）
     */
    @Bean
    public Queue orderTtlQueue() {
        return QueueBuilder
                .durable(QueueEnum.QUEUE_TTL_ORDER_CANCEL.getName())
                .withArgument("x-dead-letter-exchange", QueueEnum.QUEUE_ORDER_
                .withArgument("x-dead-letter-routing-key", QueueEnum.QUEUE_ORD
                .build();
    }

    /**
     * 将订单队列绑定到交换机
     */
    @Bean
    Binding orderBinding(DirectExchange orderDirect,Queue orderQueue){
        return BindingBuilder
                .bind(orderQueue)
                .to(orderDirect)
                .with(QueueEnum.QUEUE_ORDER_CANCEL.getRouteKey());
    }

    /**
     * 将订单延迟队列绑定到交换机
     */
    @Bean
    Binding orderTtlBinding(DirectExchange orderTtlDirect,Queue orderTtlQueue)
        return BindingBuilder
                .bind(orderTtlQueue)
                .to(orderTtlDirect)
                .with(QueueEnum.QUEUE_TTL_ORDER_CANCEL.getRouteKey());
    }

    @Bean(name = "rabbitListenerContainerFactory")
    @ConditionalOnProperty(prefix = "spring.rabbitmq.listener", name = "type",
```

```
    public SimpleRabbitListenerContainerFactory simpleRabbitListenerContainerF

        SimpleRabbitListenerContainerFactory factory = new SimpleRabbitListene
        //消息接收之前加拦截处理，每次接收消息都会调用，是有压测消息标记的，先存到副本变量，
        factory.setAfterReceivePostProcessors(new MessagePostProcessor() {
            @Override
            public Message postProcessMessage(Message message) throws AmqpExce
                Map header = message.getMessageProperties().getHeaders();
                //判断是压测的消息，动态切换影子库跟后续操作
                if (header.containsKey("test")){
                    AppContext.getContext().setFlag("7DGroup");
                    log.info("RabbitMQ 收到压测标记！");
                }
                return message;
            }
        });
        configurer.configure(factory, connectionFactory);
        return factory;
    }

`
```

这样我们的改造就完成了，接着我们来测试验证下。

第四步，接口测试验证。

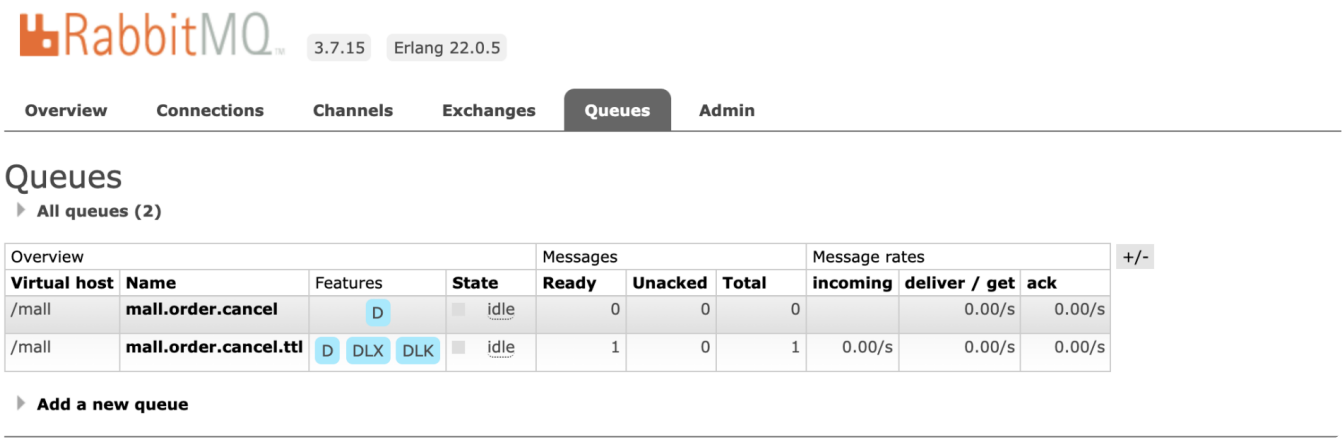
使用 curl 命令调用下单接口，这里带上了 Header 压测标记。

复制代码

```
 ~ ▶ curl -X POST -H  "Accept:*/*" -H  "Authorization:eyJhbGciOiJSUzI1NiIsInR5
{"code":200,"message":"下单成功","data":{"orderItemList":[{"orderId":31813547,"
```

注意：为了方便我们测试验证，这里已经将延迟消息时间调小了。

查看下 RabbitMQ 控制台。


RabbitMQ 3.7.15 Erlang 22.0.5
Overview Connections Channels Exchanges Queues Admin
Queues
All queues (2)
Overview Messages Message rates +/-
Virtual host Name Features State Ready Unacked Total incoming deliver / get ack
/mall mall.order.cancel D idle 0 0 0 0.00/s 0.00/s
/mall mall.order.cancel.ttl D DLX DLK idle 1 0 1 0.00/s 0.00/s 0.00/s
Add a new queue

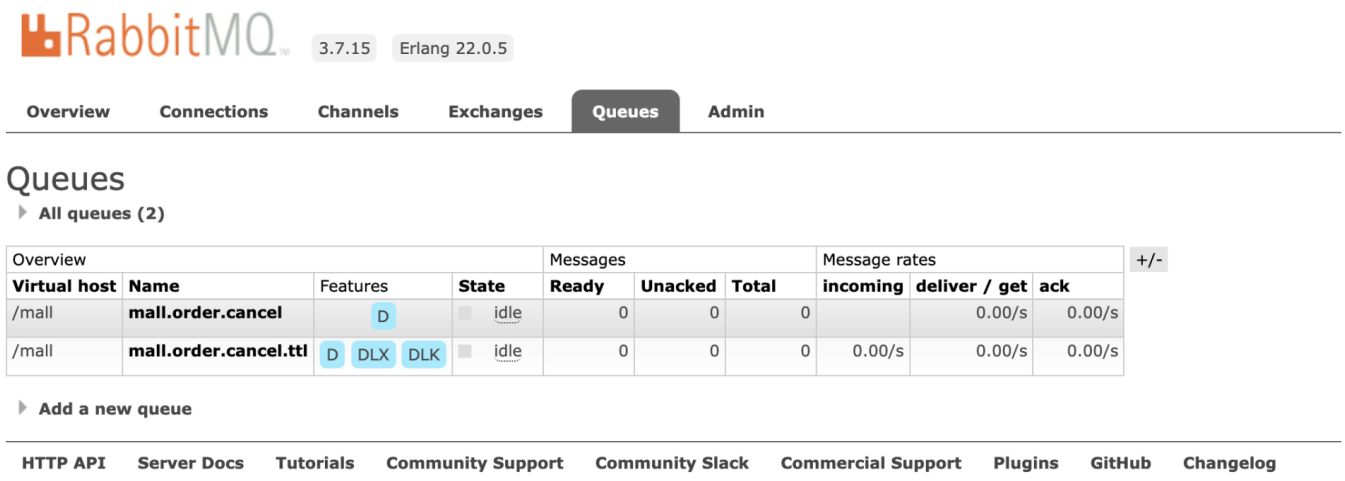
等待一会，查看下 order 服务的控制台日志。

复制代码

```
2021-10-05 17:30:30.705  INFO [mall-order,04190964bd8bce42,249e170cadf982e8,tr
2021-10-05 17:30:30.706  INFO [mall-order,04190964bd8bce42,249e170cadf982e8,tr
2021-10-05 17:30:30.706  INFO [mall-order,04190964bd8bce42,249e170cadf982e8,tr
2021-10-05 17:34:30.769  INFO [mall-order,,,] 17792 --- [ntContainer#0-1] c.du
2021-10-05 17:34:30.772  INFO [mall-order,,,] 17792 --- [ntContainer#0-1] c.d.
2021-10-05 17:34:30.902  INFO [mall-order,04190964bd8bce42,c0e11235f08391aa,tr
2021-10-05 17:34:30.991  INFO [mall-order,04190964bd8bce42,c0e11235f08391aa,tr
```

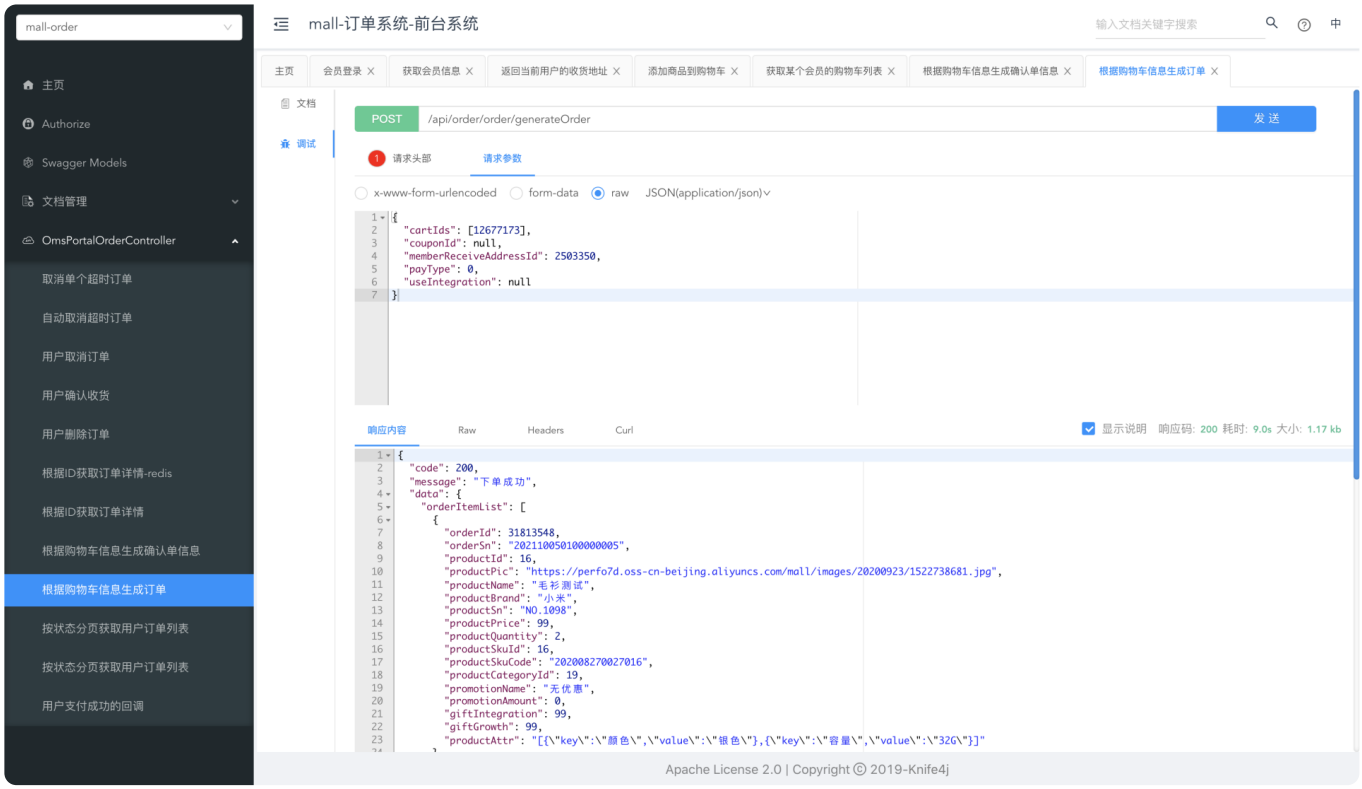
可以看到，压测标记已经被透传下去了。

再次确认下 RabbitMQ 控制台。


RabbitMQ 3.7.15 Erlang 22.0.5
Overview Connections Channels Exchanges Queues Admin
Queues
All queues (2)
Overview Messages Message rates +/-
Virtual host Name Features State Ready Unacked Total incoming deliver / get ack
/mall mall.order.cancel D idle 0 0 0 0.00/s 0.00/s
/mall mall.order.cancel.ttl D DLX DLK idle 0 0 0 0.00/s 0.00/s 0.00/s
Add a new queue
HTTP API Server Docs Tutorials Community Support Community Slack Commercial Support Plugins GitHub Changelog

截图显示，延迟的消息已经被成功消费。

接下来，我们测试一下正常请求，使用 web API 文档调用下单接口。



等待一会，查看下 Order 服务的控制台日志。

```
 18:15:58.108  INFO [mall-order,682d3874c34b8dca,f8f6916562572089,true] 18776
2021-10-05 18:15:58.114  INFO [mall-order,682d3874c34b8dca,f8f6916562572089,tr
2021-10-05 18:15:58.117  INFO [mall-order,682d3874c34b8dca,f8f6916562572089,tr
2021-10-05 18:19:58.562  INFO [mall-order,682d3874c34b8dca,f1942f9153ec7729,tr
2021-10-05 18:19:58.650  INFO [mall-order,682d3874c34b8dca,f1942f9153ec7729,tr
```

可以看到，消息是正常处理的，这说明我们的消息隔离改造成功了。

总结

好了，这节课就讲到这里。刚才，我们完整演示了 RabbitMQ 消息隔离的技术预演和真实系统改造过程。这节课有几个要点，我希望你能够记住：

1. RabbitMQ 消息隔离主要解决的是异步任务之间标记透传的需求；
2. 引入数据上下文对象能够兼容不同的应用协议，同时能够支持跨线程间及 MQ 异步任务的透传，最重要的是能支持随时存取；

3. 相比数据库隔离，消息隔离的方案较少，主要为**影子队列**和**数据偏移**。

我们这里只是介绍了 RabbitMQ 消息隔离方案。从技术实现来看，其它 MQ 产品的隔离逻辑是一致的。你也可以根据自己项目的实际情况和需求灵活选择，也欢迎有问题来和我讨论。

课后题

学完这节课，我想请你思考两个问题：

1. 除了 RabbitMQ，你在工作中还接触过哪些 MQ 产品，它们有什么区别？
2. 相比数据库隔离，你觉得 MQ 消息隔离的难点在哪里？

欢迎你在留言区与我交流讨论。我们下节课见！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 1 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 17 | 流量隔离：MongoDB 数据库隔离是怎么做的？

下一篇 19 | 日志隔离：如何落地日志隔离？

更多学习推荐

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。